第6章 組み込みCプログラミングの第一歩

── 付属基板で動作するソフトウェアの記述法

山際伸一

ここでは，組み込みCプログラミングの経験が少ない技術者を対象に，C言語によるマイコン・プログラミングの基礎を解説する．C言語によるプログラミングは大学の授業などで多少なりとも経験のある方がほとんどだと思われる．しかし組み込みプログラミングには，パソコンで動作するC言語プログラミングとは異なる独特の記述が求められる．パソコンのHello World相当の基本プログラムや，付属基板で動作する簡単なソフトウェアの記述について解説する．（編集部）

ここでは実際に手を動かしながら，組み込みプログラムをC言語で書いていきます．組み込みソフトウェア開発では，パソコン上のCプログラミングにはない独特の概念がありますが，そのような難しいことは後回しにして，まずはC言語でマイコンを制御するところに集中してみます．

付属基板には，プロセッサから制御可能なLED（D1）が搭載されています．このLEDを使って，C言語の制御コードを書いてみましょう．C言語を思い出せない読者のために簡単な文法をコラムで解説しますので，随時参考にしてください．

演習1：LEDを点灯させる

まず，LEDを制御するもっとも簡単なコードを書いてみましょう．LEDを点灯させるだけのC言語プログラムを**リスト1**に示します．

● LEDの転送手順を確認する

LED（D1）はP4.7に接続されています．そしてレジスタのGP4CONとGP4DATを操作することでGPIOを制御します．LEDはI/Oが“L”出力の時に点灯するように接続されています．

LEDを点灯させる手順を次に示します．

6

リスト1
practice1.c

```
int main(){
  // MMRへのポインタ
  volatile unsigned int *mmr_base;                          ┐
  // GPIO設定レジスタへのポインタ                              ├── ①
  volatile unsigned int *gp4con, *gp4dat;                   ┘

  // MMRは0xFFFF0000番地                                     ┐
  mmr_base = (unsigned int *)0xFFFF0000;                     │
                                                             │
  // MMRと同様にGPIOのレジスタを設定                           ├── ②
  gp4con = mmr_base + (0xF410/sizeof(unsigned int *));       │
  gp4dat = mmr_base + (0xF460/sizeof(unsigned int *));       ┘

  // GP4CONレジスタでGPIOのP4.7をGPIOとして設定
  *gp4con = 0x00000000;
  // GP4DATレジスタでP4.7を0を設定（read onlyの下位16ビットは何を書いてもよい）
  *gp4dat = 0xFF7F0000;   ◀── ③

  while(1){}; // リターンする場所がないのでここで無限ループにはめる   ◀── ④

  return 0;
}
```

- GP4CONレジスタのP4.7に関するビット（ビット29-28）に0を設定
- GP4DATレジスタのP4.7に関する方向ビット（ビット31）に1を設定
- GP4DATレジスタのP4.7に関する出力値ビット（ビット23）に0を設定（“L”を出力する設定）

この手順をC言語で記述したのが**リスト1**ということになります．`int`型を返す`main`という関数（左掲のコラム「いまさら聞けないC言語1――変数の型」と下掲のコラム「いまさら聞けないC言語2――関数」を参照）で，LEDを点灯するまでのすべての処理を行います．

Column

いまさら聞けないC言語1――変数の型

C言語の変数の型は，`int`，`short`，`char`がもっとも一般的に使われます．符号なしの変数の場合，これらの前に`unsigned`が用いられます．代表的な型を**表A-1**に示します．

変数の型の使い分けとしてたいせつなことは，符号の有無です．例えば，ハードウェアの1ビット・ラインを制御する場合などは，符号は関係ないので`unsigned`を用います．

また変数の大きさも重要です．32ビット・プロセッサでは，32ビットが1ワードになります．`int`型はGCCでは，ワードの長さに割り当てられるので，32ビット・プロセッサでは，`int`型は4バイトの大きさということになります．`short`はその半分，`char`はさらにその半分となり，それぞれ2バイト，1バイトとなります．

表A-1　代表的な型

型	意　味
int	符号付き整数
unsigned int	符号なし整数
short	符号付きハーフワード
unsigned short	符号なしハーフワード
char	符号付きキャラクタ
unsigned char	符号なしキャラクタ

● レジスタへのポインタ宣言とvolatile

リスト1の①の変数宣言部では，レジスタ群のベース・アドレスとGP4CONとGP4DATの二つのレジスタのアドレスを保持するためのポインタを宣言します（右掲のコラム「いまさら聞けないC言語3――ポインタ」を参照）．

このポインタを宣言しているところに`volatile`という見慣れぬキーワードがあると思います．これは，組み込みプログラム特有のC言語文法といえます．組み込みではない場合でも時折，同様の目的で用いることはありますが，組み込みソフトウェアでは必須のキーワードになります．

`volatile`という英語は「揮発性の，気まぐれな」という意味です．つまり，ここで宣言されたポインタの指す先のデータは「気まぐれに揮発」してしまう可能性があるのです．したがって，このポインタをもとに読み出すデータは，読み出すごとに値が異なる可能性があるということです．書き込みに関しても同様で，書き込んだ値はかならずしも永久に保存されているとは限らないということです．

これは，マイコン外部のハードウェアの制御を例にするとわかりやすく説明できます．例えば，通信コントローラがマイコンの外部に接続されているとします．その通信コントローラには，データ受信フラグのビットがあるとします．通信コントローラのデータ受信フラグは，‘1’が‘0’に固定されるわけではありません．受信パケットの状態やコ

Column

いまさら聞けないC言語2――関数

C言語では，処理のまとまりを関数で記述します．関数の構造を**図B-1**に示します．

`main`という名まえが一般的に使われ，C言語の教科書の多くは，その関数が最初に実行されると解説されています．しかし，実際にはそれは厳密な解説ではありません．コンパイラが最初に実行する関数を`main`という名まえの関数と規定しているだけで，実は，どんな関数から始めてもよいのです．

```
戻り値の型　関数の名まえ（引き数の型0 引き数0，引き数の型1 引き数1，…，引き数の型N 引き数N）{
        変数宣言部
        本体

        return 返値；
}
```

図B-1 関数の構造

ントローラの処理に依存して変化するものです．

volatileを付けない場合，つまり，

```
unsigned int *mmr_base;
unsigned int *gp4con, *gp4dat;
```

のようにしてしまうと，ここで宣言されたポインタの指す先のデータは揮発しないものとして扱われます．コンパイラは，I/Oレジスタの読み出しではなく，CPU内部レジスタだけで処理するように最適化処理を施してしまう可能性があります．これらのポインタの指す先を何度アクセスしても，実際のポインタの指す先がアクセスされるのは最初の1回だけになってしまうということです．前述の通信コントローラの例であれば，受信フラグは最初のアクセス以降，まったく変化しないことになります．

ポインタの指す先へのアクセスをきちんと反映させるためには，volatileキーワードを忘れずに付けることがたいせつです．

● 型宣言とレジスタ長

変数宣言では，unsigned int *を用いています．

MMRの各レジスタは，32ビット幅であり，4バイトごとのアドレスに配置されています．4バイトの変数，つまり，整数型を使うわけです．

unsignedを指定しているのは，レジスタに符号が存在しないためです．レジスタには符号なしの変数を用います．

● ポインタへのアドレス設定

宣言されたポインタへ実際のアドレスを設定しているのは，**リスト1**の②の部分です．

まず，MMRのベース・アドレスを，

```
mmr_base =(unsigned int *)
                         0xFFFF0000;
```

として，mmr_baseポインタに0xFFFF0000を代入しています．

ポインタは，データ本体へのアドレスです．したがってこの場合，MMR領域の先頭アドレスをmmr_baseに代入しています．この際には，アドレス値はmmr_baseの型に合わせる必要があります．(unsigned int *)という記述は，定数値の型を指定するためのものです．これをキャストと呼びます．定数値を記述するだけでは，型がわかりません．そこでC言語の仕様では，代入先の変数型にキャ

Column いまさら聞けないC言語3 ――ポインタ

ポインタは，C言語を勉強する人をもっとも苦しめる概念になっているようです．しかし，組み込みソフトウェアでは，あたりまえのように使われています．ハードウェア制御にとてもマッチするしくみだからです．

ポインタは，「データのありかを指すアドレス」です．例えば，ポインタをpとすると，

```
unsigned int *p;
```

のように「*」を変数名の前に付けて宣言します．これはつまり，アドレスを入れる箱pができたことだと考えてください．

このpにアドレスを入れてみます．

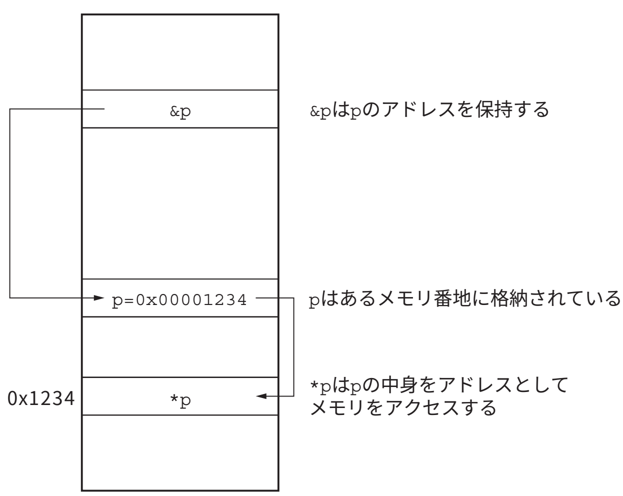
```
p =(unsigned int *)0x00001234;
```

箱に0x00001234という値が入りました．この状態を「ポインタpは0x00001234番地を指す」といいます．

0x00001234番地の内容を参照する際には*pと書きます．

図C-1にこのようすを示します．

ポインタのポインタというのもよくあります．&pと書いた場合，ポインタのポインタということになります．この場合，pというポインタの中身(つまり，0x00001234)を指すためのポインタということになります．したがって，&pは，pがあるメモリ・アドレスということになります．



図C-1 ポインタ操作

ストすることを規定しています．

● ポインタの計算

gp4conとgp4datの二つのポインタは，MMRのベース・アドレスからのオフセット値で指定します．ひとまとまりのレジスタ群にポインタを設定する場合，ベース・アドレス＋オフセットとしたほうが保守性が高まります．例えば，レジスタ群がまとめてアドレス変更になった場合，ベース・アドレスだけを修正すればすみます．レジスタどうしの関係も明確になります．

GP4CONレジスタは，MMRのベース・アドレス（0xFFFF0000）からオフセット0xF410のところにあります．すなわち，0xFFFF0000＋0xF410番地です．そこで，gp4conにポインタを設定する際に，

```
gp4con = mmr_base + 0xF410;
```

としたいところですが，これは大きな誤りです．C言語の仕様では，この一文を「mmr_baseポインタに，mmr_baseの型の大きさを0xF410個分増加させた値をgp4conに代入する」と解釈します．ここでmmr_baseはunsigned int型へのポインタです．unsigned int *型は，ARMプロセッサでは4バイトです．したがって，mmr_base＋0xF410は，0xFFFF0000＋4×0xF410となってしまいます．0xF410の1/4の値をmmr_baseに加えれば，正しい値になります．

```
gp4con = mmr_base + 0xF410/4 ;
```

しかし，もしunsigned int *型が4バイトの大きさでなくなった場合，例えば64ビットCPUに移植された場合に正しく動作しません．そこで，**リスト1**の②では，

```
gp4con = mmr_base
        +(0xF410/sizeof(unsigned int *));
```

と記述しています．sizeof演算子は，unsigned int *型の大きさを返す演算子です．

● MMRへの書き込み

ポインタのアドレス設定ができたところで，レジスタ値を変更できるようになります．

リスト1の③で値を書き込んで，LEDの点灯を行っています．

● プログラムの最後で気をつけること

リスト1の④は，while文による無限ループです（右掲のコラム「いまさら聞けないC言語5 ── ループ」を参照）．

OSの上で動作するソフトウェアの場合は，OSに戻ることになりますが，ここで動作するソフトウェアはreturnによって戻る場所がありません．戻るところのない組み込みソフトウェアでは，終わらないしくみにしなくてはなりません．

● スタートアップ・ルーチン

LEDを点灯させる処理はすべて**リスト1**に記述されています．しかし，これだけでは実際の組み込みシステムで動

Column　いまさら聞けないC言語4 ── 演算子

C言語の演算子はたくさんあります．その中でも，よく用いる演算子を列挙してみます．

まず，四則演算（+，-，*，/）です．数値演算に使われます．論理演算としてはAND（&），OR（|），XOR（^），NOT（~）があります．NOTだけは特殊で，反転する変数または数値の前に置きます．例えば，~0xFFFEとすると0x0001が得られます．

シフト演算子として，右シフト（>>）と左シフト（<<）があります．

これらの演算子には優先順位があります．例えば，四則演算については，*と/は+と-よりも先に演算されます．演算結果がおかしいときは，優先度を疑ってみるといいと思います．

C言語でも優先順位の低い演算を先に実行したいときには，かっこでくくります．例えば，a+bの結果にcを掛ける場合は，

```
d =(a + b)*c;
```

と記述します．演算子の優先順位に頼らず，かっこを使うことで演算の意図を明確にすることができ，保守性も高まります．

これらの演算子のほかに，型の大きさを求めるsizeof演算子があります．sizeofは関数のように使われ，型や変数の大きさを測ることができます．例えば，

```
unsigned int a;
```

と宣言されているとき，sizeof(a)とすると，32ビット・プロセッサでは4と返されます．

作しません．マイコンやシステムの初期化などを行うスタートアップ・ルーチンが必要になります．

マイコンは，電源がONになるか，リセットがかかった後，0x00000000番地（またはマイコンごとに決められている実行開始アドレス）から命令を実行し始めます．しかし，このときのレジスタなどの状態は，すべてあらかじめ決められているリセット値です．

ソフトウェアの動作には，例えば作業用メモリ領域（スタック）が必要です．関数の引き数と戻り値，戻り先アドレスなどの保存や，限られた演算レジスタで複雑な処理を行うために，データを一時待避するために使われます．マイコンにはこの作業領域を示すポインタ（スタック・ポインタ）がありますが，システムごとに適切なアドレスを設定する必要があります．Cをはじめとする高級言語では，本来は意識しなくてすむものですが，組み込みソフトウェアでは忘れてはならない重要な処理となります．

スタック・ポインタが正しく設定されていないと，ソフトウェアは正しく動作しません．関数を呼び出す前に設定が必要です．すなわち，main関数の実行の前にきちんと設定しないといけません．つまり，C言語で記述できないということになります．そこでアセンブリ言語で記述します．このアセンブリ言語で書かれたコードをスタートアップ・ルーチンと呼びます．

リスト2に，**リスト1**の実行に必要なスタートアップ・ルーチンを示します．ここでは最低限必要なスタック・ポインタの設定と，Cのmainへの分岐だけを行っています．

Column いまさら聞けないC言語5 ──ループ

ループとは同じコードを繰り返し実行する文です．C言語のループにはfor文とwhile文があります．

● for文

for文はインデックスと呼ばれる変数と条件で繰り返し回数を制限します．文法を**図E-1(a)**に示します．

例えば，10回繰り返すコードであれば，**図E-1(b)**のように記述します．

インデックスの初期値，条件，インデックスの繰り返しごとの変更は省略も可能で，

```
for(;;){}
```

と書くことで無限ループを作ることができます．

for文の特徴としては，インデックスの初期値が最初に実行されて，ループに突入していくことです．はじめから条件に合わない場合には，繰り返されるコードは実行されません．

● while文

while文の文法を**図E-2(a)**に示します．2種類の書きかたがあります．

図E-2(a)では，繰り返されるコードの前に条件判断が実行されます．この記述を使って，**図E-1(b)**の10回繰り返すfor文と等価なwhile文を書いたのが**図E-2(b)**です．iをインクリメントする処理がwhile文の中にあることに注意してください．

繰り返しコードをかならず1回実行してから繰り返し判定をさせたい場合は，**図E-2(c)**の文法を使います．文の最初はdoから始まります．この場合，繰り返されるコードを実行した後で条件を判定するので，有無を言わせず一度は繰り返されるコードが実行されることになります．

```
for（インデックスの初期値;条件;インデックスの繰り返しごとの変更）{
        繰り返すコード
}
```

(a) 文法

```
int i;

for(i=0;i<10;i++){
        ...
}
```

(b) 記述例

図E-1 for文の文法

```
while(条件){
        繰り返されるコード
}
```

(a) 文法1

```
int i;
i = 0;
while(i<10){
        繰り返されるコード
        i++;
}
```

(b) 記述例

```
do{
        繰り返されるコード
}while(条件);
```

(c) 文法2

図E-2 while文の文法

リスト2　startup.s

```
      .text

      .extern            main
      .extern            _sp_base
#=================================================
#      Initialize vectors
#
##     0x00     Reset
##     0x04     Undefined Instruction
##     0x08     Software Interrupt
##     0x0c     Abort(prefetch)
##     0x10     Abort(data)
##     0x14     Reserved
##     0x18     IRQ
##     0x1c     FIQ
#=================================================

      B         _startup

      .org 0x20
_startup:
      LDR       R13,=_sp_base
      BL        main
```

マイコンにリセットがかかると，

```
B _startup
```

を実行します．`B`命令により，`_startup`というラベルの付く番地にジャンプします．

`_startup`というラベルから，初期設定の記述があります．

```
LDR R13,=_sp_base
```

でR13（スタック・ポインタ）にスタック領域のアドレスを入れます．`_sp_base`がスタック領域のアドレスです．そして，

```
BL main
```

で`main`関数へと分岐しています．`BL`命令はリンク付き分岐命令で，リンク・レジスタにこの命令の番地に戻るためのアドレスを入れて，C言語で記述された`main`関数へと分岐していきます．

C言語記述のはずがARMのアセンブリ言語記述が出てきてしまい，たいへんなことになってきたと思われるかもしれません．安心してください．本稿の範囲では，**リスト2**をそのまま使うことができるので，スタートアップ・ルーチンというものが必要だという程度に知っておいてもらえればいいのです．マイコンの機能を使いこなそうとすると，スタートアップ・ルーチンの処理も増えてきます．しかし，統合開発環境の中には，スタートアップ・ルーチンを自動生成してくれるものもあります．

リスト3　memory.def

```
SECTIONS
{
      .start 0x00000000 : {
      _sstarttext = .;
               startup.o(.text)
      _estarttext = .;
      }

      .text : {
      _stext = .;
               *(.text)
      _etext = .;
      }

      .rdata : {
      _srdata = .;
               *(.rodata)
             *(.rodata.str1.4)
      _erdata = .;
      }

      .data 0x00010000: {
      _sdata = .;
               *(.data)
               *(.zdata)
      _edata = .;
      }
      . = ALIGN(0x4);
      _end = .;
      . += 0x100;
      _sp_base = .;
}
```

● コンパイルと実行

GCCを使うコンパイル手順を以下に示します．コンパイルの際には，メモリ配置を定義するmemory.def（**リスト3**）というファイルを使用します．ARM開発環境のコマンドへのパスは設定されているものとします．

```
$ arm-elf-gcc -c practice1.c
$ arm-elf-as startup.s -o startup.o
$ arm-elf-ld -T memory.def startup.
             o practice1.o -o practice1
$ arm-elf-objcopy -o ihex practice1
                              practice1.hey
```

これでIntel HEX形式のオブジェクト・ファイルができました．これをダウンロードし，リセットして（SW1を押して）からプログラムを実行するとLED（D1）が点灯します．

演習2：LEDを点滅させる

LEDを点灯できたら，この応用として点滅させることにしましょう．

LEDを点滅させるには，ONとOFFを一定期間ごとに

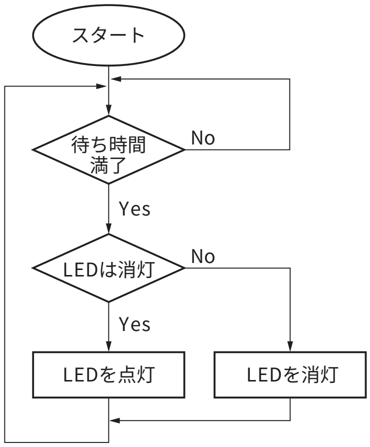


図1　LEDをチカチカさせる無限ループ

リスト4　practice2.c

```
#define ITERATION 200000

int main(){
  // MMRへのポインタ
  volatile register unsigned int *mmr_base;
  // GPIO設定レジスタへのポインタ
  volatile register unsigned int *gp4con, *gp4dat, *gp4set;
  register int i;
  register int led_state; // 1のとき点灯，0のとき消灯
  // MMRは0xFFFF0000番地
  mmr_base = (unsigned int *)0xFFFF0000;

  // MMRと同様にGPIOのレジスタを設定
  gp4con = mmr_base + (0xF410/sizeof(unsigned int *));
  gp4dat = mmr_base + (0xF460/sizeof(unsigned int *));        ← ①

  // GP4CONレジスタでGPIOのP4.7をGPIOとして設定
  *gp4con = 0x00000000;                                       ← ②

  *gp4dat = 0xFF7F0000; // 点灯させる
  led_state = 1;

  i = 0;
  while(1){ // リターンする場所がないのでここで無限ループにはめる ← ③
    if(i>ITERATION){
      i= 0;
      // GP4DATレジスタでP4.7を反転させる（read onlyの下位16ビットは何を書いてもよい）
      if(led_state == 0){ // 消灯
      *gp4dat = 0xFF7F0000; // 点灯させる
      led_state = 1;
      }
      else{ // 点灯
      *gp4dat = 0xFFFF0000; // 消灯させる                     ← ④
      led_state = 0;
      }
    }
    i++;
  }

  return 0;
}
```

繰り返せばよいわけです．具体的には，**図1**のような無限ループを記述します．

LEDを点滅させるCソース・コードを**リスト4**に示します．

● 変数宣言

リスト4の①はポインタ宣言とその初期化です．この部分は，LEDを点灯させたときとほぼ同じものです．`i`は後に使う待ち時間をカウントするための変数，`led_state`はLEDの状態を表すための変数です．

ここで，`register`というキーワードが出てきています．`register`は変数宣言の前に記述することで，その変数をレジスタに割り当てるように制御するためのものです．`register`と指定しない場合には，メモリに配置されてしまうかもしれません．

`register`を付けると変数が可能な限りレジスタに配置されるようにコンパイラは処理するので，処理の高速化に有効な場合があります．

ここで用いる`led_state`と`i`は`volatile`宣言を必要としません．`i`と`led_state`は，`volatile`で宣言されている`gp0dat`の順序を制御している`if`文で使われていることにより，かならず評価されるためです．

● MMRへの書き込み

リスト4の②ではGP4CONを設定し，P4.7をGPIOに設定しています．

● ループ

リスト4の③の`while`文は，無限ループ（条件は1なので，つねに真となって繰り返される）となっています．つまり最後の`return`には行き着かずに，そのループの中身を永久に実行し続けるのです．

`while`文の中では，`i`が`ITERATION`より大きくなった場合に，`if`文を用いてLEDの状態を検査します．そして，

消灯している(led_stateが0)ときには点灯させるため，led_stateに1を代入します．また，点灯しているときには消灯させるため，led_stateに0を代入しています．

for文はITERATIONの回数だけループを繰り返しています．if文を抜けたところでiを1だけ増加させているので，このwhile文の中身をITERATION回実行し，LEDの状態を変更しているわけです．したがって，LEDの状態変化の後，待ち時間が発生し，待ち時間に達するとLEDの状態を反転させるように動作します．

繰り返し回数であるITERATIONは，main関数の中で宣言されていません．実はこれは定数で，いちばん上にある，

```
#define ITERATION 200000
```

で定義されています．このdefineをC言語の世界ではマクロと呼びます．この1文で「ITERATIONという文字列は200000という文字列と置き換える」ことを定義しています．ですので，このfor文は，

```
for(i=0;i<200000;i++){}
```

という意味になります．

ITERATIONは変数ではありません．あくまでも文字列の変換される定義です．したがって，ITERATIONへの代入，つまり，

```
ITERATION = 10;
```

といったコードはエラーになります．

● LEDの点滅

リスト4の④は，LEDを反転する処理です．つまり，led_stateが0のときは消灯ですから，点灯するように，GP4DATに値を書き込んでいます．その逆の場合は，GP4DATの23ビット目(P4.7の場所)に1を書き込んでいます．

● コンパイルと実行

GCCによるコンパイル手順を示します．演習1で用いたstartup.s(**リスト2**)とmemory.def(**リスト3**)が必要です．

```
$ arm-elf-gcc -c practice2.c
$ arm-elf-as startup.s -o startup.o
$ arm-elf-ld -T  memory.def
     startup.o practice1.o -o practice2
$ arm-elf-objcopy -O ihex practice1
                              practice1.hex
```

生成されたHEXファイルをダウンロードすると，定期的にD1が点滅することが確認できます．

まとめ

C言語の基礎を振り返りながら，組み込みソフトウェアの最低限の記述を説明しました．組み込みソフトウェア開発では，システムの動作についての理解と，その動作をこと細かに記述するC言語文法の理解が必要になります．

LEDを点滅させるだけでも，知らなければならないことがたくさんありました．しかも本稿は，付属基板を動作させるに当たっての最低限の説明だけにとどめています．より高度な処理の記述やマイコンの制御方法については，稿をあらためて解説したいと思います．

参考・引用*文献

(1) Inerface編集部 編；ARMプロセッサ入門，TECH-I Vol.18，CQ出版社．
(2) 特集 フリーソフトウェア活用組み込みプログラミング，Interface，2005年1月号，pp.43-134.

やまぎわ・しんいち
工学博士